

# Parcial de Organización del Computador I

Ramiro Basile - 108/20

Primer cuatrimestre de 2021

## Ejercicio 1

**RISC-V maneja el principio de simplicidad, en relación a esto responda:**

**¿Acceder a un operando en registro es más rápido que buscar el operando en memoria?**

Los registros son más rápidos de leer y escribir que la memoria. Esto se debe a que, más allá de cualquier diferencia en la tecnología que usen, la memoria se accede a través de algún tipo de controlador que selecciona una posición dada una dirección, a diferencia de los registros que van cada uno directo al bus.

Esto es exactamente lo que pasa en la máquina Orga1, donde los registros simplemente pasan por un multiplexor directamente al bus, mientras que la memoria debe ser leída de RAM.

**A partir del inciso anterior, ¿cómo cree que impacta al rendimiento del programa y a la arquitectura la cantidad de registros disponibles?**

Generalmente la mejor forma de guardar un dato suele ser en un registro, por esa ventaja de velocidad mencionada anteriormente. Datos temporales que se pueden sobrescribir sin consecuencia después de usarse o datos que se van a acceder muy frecuentemente son buenos candidatos para ser guardados en un registro en vez de memoria. Si a la hora de guardar datos el programador (o el compilador) tiene varios registros a su disposición, el rendimiento del programa puede mejorar muchísimo.

Sin embargo, el tamaño de las instrucciones se ve afectado por la cantidad de registros en la ISA, ya que se requieren  $2^{\#registros}$  bits de una instrucción por cada registro que direcciona la instrucción. Los 32 registros que especifica RV32I (pág. 21), por ejemplo, demandan 5 bits. La máquina Orga1, por dar otro ejemplo, requiere 3.

También, en general, la velocidad de entrar y salir de subrutinas puede verse afectada por la cantidad de instrucciones, ya que varios de (o todos) los registros se van a tener que guardar y recuperar del stack al entrar y salir de ellas—y a

más registros que se tengan que guardar y recuperar, más lento se vuelve entrar y salir de la subrutina.

La conclusión que se puede sacar de esto es que una ISA tiene que optar en su diseño por un cierto balance entre el mayor rendimiento que permite una buena cantidad de registros y el largo de instrucción y menor velocidad de subrutinas que conlleva.

RISC-V, en particular, aprovecha su generosa cantidad de registros no solo para ofrecerle a los programadores (o el compilador) más registros, sino también para aplicar una filosofía que simplifica bastante la ISA: todas las operaciones aritméticas se realizan entre registros (pág. 25). Acá la idea es que, en vez de darle a cada operación varias instrucciones distintas—una registro-registro, otra memoria-registro, otra registro-memoria, etc.—solo existe una única instrucción que opera entre registros (la forma más eficiente de hacerlo) y se espera que se acceda a memoria solo cuando es necesario y usando alguna de las pocas instrucciones de tipo *load y store*. Esto mantiene el ISA simple, con menos instrucciones y formatos, pero no menos versátil o más compleja.

Y en el frente de la velocidad de entrada y salida de subrutinas, RISC-V especifica una *convención de llamada* que mitiga en cierta medida el costo que puede llegar a tener el tener muchos registros. Esta convención define ciertos registros que persisten a través de llamadas a subrutinas (*saved*) y otros que no necesariamente se preservan (*temporary*) (pág. 34). Lo que logra esto es que subrutinas no-recursivas que precisen pocos registros pueden no tocar ningún registro *saved* y así no tener que guardar nada en el stack, ahorrando lentas lecturas y escrituras de memoria innecesarias y así agilizando las subrutinas (pág. 35).

## Ejercicio 2

¿Cuál es la ventaja de tener un registro de valor constante 0(x0)?

RV32I solo especifica 47 instrucciones (pág. i) y 6 formatos (pág. 17) para mantener la ISA simple y elegante. Para no pagar esta simplicidad con expresividad a la hora de programar, RISC-V usa de forma muy ingeniosa el registro x0 para proveer *pseudoinstrucciones*, que funcionan como “alias” de instrucciones reales, pero omitiendo o reutilizando los campos de la instrucción original no necesitan con pasarles el registro x0 para ignorar escrituras o fijar un 0 en comparaciones u otras operaciones (pág. 37).

Un ejemplo de esto podría ser `neg rd, rs`, que no es en realidad una instrucción sino un alias a `sub rd, x0, rs`—acá la resta del registro a 0 resulta en la negación del registro. Otro buen ejemplo es la serie de saltos condicionales respecto a 0 (`beqz`, `bnez`, `blez`, etc.) que funcionan simplemente usando su respectiva contra-parte en las instrucciones base (`beq`, `bne`, `ble`, etc.) pero con x0 como el segundo operando para poder comparar con 0.

Si bien no todas las pseudoinstrucciones dependen del registro `x0`, 32 de ellas lo hacen (pág. 38), para darse una idea lo crucial que es.

Estas pseudoinstrucciones no están presentes en la máquina Orga1 ni tampoco un registro que apunte permanentemente a 0.

**¿Cómo maneja las escrituras a este registro y por qué lo hace de esa forma?**

No lo hace, la escritura no va a ningún lado (como `/dev/null` en UNIX) y la lectura simplemente lee una constante 0 (pág. 20). Justamente por esto (pseudo)instrucciones como `j` y `ret` pasan `x0` por `rd`, ya que el valor que quedaría en `rd` no cumpliría ninguna función en el contexto de un salto o un return.

## Ejercicio 3

**¿Cómo se resuelve el overflow?**

Las instrucciones de RISC-V descartan el overflow aritmético, ya que se considera que no suele ser necesario detectarlo en la mayoría de programas y, en el caso de que lo fuera, es fácil de obtener haciendo algunas operaciones adicionales (pág. 24).

Detectar el overflow de suma sin signos es trivial: solo hace falta hacer un `bltu` compare el resultado con el primer operando y será overflow si se cumple:

```
addu t0, t1, t2
bltu t0, t1, overflow
```

```
overflow:
```

```
...
```

El caso del overflow de suma sin signo requiere algunas instrucciones más:

```
add t0, t1, t2
slti t3, t2, 0          # t3 = (t2 < 0)
slt t4, t0, t1          # t4 = (t1 + t2 < t1)
bne t3, t4, overflow    # overflow si (t2 < 0) && (t1 + t2 >= t1)
                        # || (t2 >= 0) && (t1 + t2 < t1)
```

```
overflow:
```

```
...
```

La máquina Orga1, a diferencia, deja que el overflow suceda y levanta un flag de overflow para indicar que la operación resultó en overflow. Esto se paga con mayor complejidad a nivel hardware: la máquina Orga1 tiene un componente en la ALU que funciona con la misma lógica del código anterior.

## Ejercicio 4

### ¿Cómo se resuelve la lógica de control (branching)?

La lógica de control en RISC-V se realiza con de instrucciones de tipo jump and link (`jal`, `jalr`) o condicionalmente con instrucciones de tipo branch (`beq`, `bne`, `blt`, etc.). Para realizar un salto, estas instrucciones toman un “offset”—un desplazamiento relativo al PC (al que se le aplica `sigext` y, en `jalr`, se le suma el valor de un registro)—cuyo largo depende de si es de tipo jump and link o branch, los jump and link teniendo más bits disponibles en la instrucción que los branch. Los jump and link también guardan el valor original del PC en un `rd`, de ahí la parte de “link”.

### ¿Qué similitudes y/o diferencias existen con la máquina Orga1?

Funciona similarmente en el sentido de que los saltos son relativos, pero las instrucciones en sí son distintas. Los saltos condicionales de la máquina Orga1 no deciden en base de los flags que levantó la ALU de una operación anterior, sino que la instrucción en sí realiza la operación internamente.

## Ejercicio 5

### ¿Qué significa *position independent code*?

Un programa esta hecho con PIC cuando funciona independientemente de la posición en la que fue cargado en memoria. RISC-V se presta muy bien a escribir PIC ya que su branching es relativo al PC y su filosofía de operar entre registros minimiza la lectura y escritura a memoria de forma directa.

### ¿Qué ventaja tiene sobre el código dependiente de posición?

Los programas generalmente se modularizan para poder compilar únicamente partes de él que cambian (pág. 41). Luego de la compilación los une un linker. Por ejemplo, las librerías que podría llegar a necesitar un programa probablemente ya estén compiladas, sería una perdida de tiempo compilarlas cada vez que cambie el programa.

Linkear, sin embargo, es complicado. RISC-V especifica en que partes de la memoria se almacena cada cosa—números, strings, elementos del stack, etc—y el linker tiene que asegurarse de que las posiciones que referencia cada programa siendo linkeado conformen con esta especificación, modificando a que posiciones apunta cada instrucción del programa. Esto resulta mucho más fácil cuando se tiene PIC, ya la mayoría del trabajo del linker ya esta hecho.

## Ejercicio 6

¿En qué posición dentro de la instrucción se encuentran los bits de los registros destino y origen? ¿Depende del tipo de instrucción o de

### la instrucción en sí?

Depende del formato de la instrucción que está indicado en los primeros 6 bits de cualquier instrucción (**opcode**). Todos los formatos de instrucción que toman un registro destino **rd** lo hacen de bits 6 a 11, luego el primer registro de origen **rs1** de bits 15 a 19 y el segundo **rs2** de bits 19 a 24. Esto no quiere decir que todos los formatos tomen un registro de destino y dos de origen, sino que todos los registros están en posiciones predecibles si es que el formato los incluye (pág. 17).

### ¿Por qué fue diseñado así el formato de instrucción?

Que todas las instrucciones que toman un **rd**, **rd** y **rs1** y **rd**, **rs1** y **rs2** en los mismos bits que las otras permite que se comience el acceso a esos registros antes de la decodificación, ya que aún sin saber exactamente cual instrucción se va a ejecutar, se puede saber que los bits 6 a 11 apuntan a un registro si la instrucción toma un **rd** y ya empezar a accederlo—y así con todos los otros registros (pág. 19).

---

Referencia: *Guía Práctica de RISC-V: El Atlas de una Arquitectura Abierta*